PeerCube: an Hypercube-based P2P Overlay Robust against Collusion and Churn

Emmanuelle Anceaume, Francisco Brasileiro, Romaric Ludinard, Aina Ravoaja

I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1888

PEERCUBE: AN HYPERCUBE-BASED P2P OVERLAY ROBUST AGAINST COLLUSION AND CHURN

EMANUELLE ANCEAUME, FRANCISCO BRASILEIRO,
ROMARIC LUDINARD, AINA RAVOAJA

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

PeerCube: an Hypercube-based P2P Overlay Robust against Collusion and Churn

Emanuelle Anceaume[*] , Francisco Brasileiro[**] , Romaric Ludinard[***] , Aina Ravoaja[****]

Systèmes communicants
Projet ADEPT

Publication interne n ° 1888 — february 2008 — 28 pages

Abstract: In this paper we present PeerCube, a DHT-based system that aims at minimizing performance penalties caused by high churn while preventing malicious peers from subverting the system through collusion. This is achieved by *i)* applying a clustering strategy to support quorum-based operations; *ii)* using a randomised insertion algorithm to reduce the probability with which colluding Byzantine peers corrupt clusters, and; *iii)* leveraging on the properties of PeerCube's hypercube structure to allow operations to be successfully handled despite the corruption of some clusters. Despite a powerful adversary that can inspect the whole system and issue malicious join requests as often as it wishes, PeerCube guarantees robust operations in $\mathcal{O}(logN)$ messages, with N the number of peers in the system. Extended simulations validate PeerCube robustness.

Key-words: P2P system, Byzantine failure model, Collusion, Churn, simulation

(Résumé : tsvp)

[*] IRISA/CNRS, Campus Universitaire de Beaulieu, Rennes, France, {anceaume,roludina,aravoaja}@irisa.fr
[**] Universidade Federal de Campina Grande, Laboratório de Sistemas Distribuídos, 58.109-970, Campina Grande, PB, Brazil, fubica@dsc.ufcg.edu.br
[***] IRISA/INRIA, Campus Universitaire de Beaulieu, Rennes, France, roludina@irisa.fr
[****] IRISA/ENS Cachan, Campus Universitaire de Beaulieu, Rennes, France, aravoaja@irisa.fr

PeerCube: Infrastructure logique robuste aux collusions et forte dynamicité

Résumé : Dans ce rapport nous proposons une infrastructure pour systèmes pair-à-pair fortement dynamiques robuste aux intrusions coordonnées

Mots clés : système pair-à-pair, fautes byzantines, collusion, dynamique, simulation

1 Introduction

Research on the development of efficient peer-to-peer systems has recently received a lot of attention. This has led to the construction of numerous structured peer-to-peer overlays systems (e.g. CAN [15], Chord [23], Pastry [18], Tapestry [9] to cite some of them). All these systems are based on distributed hash tables (DHTs) which partition an identifier space among all the peers of the system. Structured overlays enjoy numerous important properties. They are efficient, scalable, and tolerant to benign failures. However, less investigation has been carried out for handling both very high churn and collusive behaviour issues. As pointed out by Locher et al. [12], most proposed peer-to-peer overlays are highly satisfactory in terms of efficiency, scalability and fault tolerance when evolving in weakly dynamic environments. On the other hand, in the presence of very frequent connections/disconnections of peers, a very large number of join and leave operations are locally triggered engendering accordingly multiple and concurrent maintenance traffic. Ensuring routing tables consistency quickly becomes unbearable, leading to misrouting, and to possible partitioning of the system. The other fundamental issue faced by any practical open system is the inevitable presence of malicious peers [21]. Guaranteeing the liveness of these systems requires their ability to self-heal or at least to self-protect against this adversity. Malicious peers can devise complex strategies to prevent peers from discovering the correct mapping between peers and data keys. They can mount *Sybil attacks* [6] (i.e., an attacker generates numerous fake peers to pollute the system), they can do *routing-table poisoning* (also called *eclipse attacks* [3, 21]) by having good peers redirecting outgoing links towards malicious ones, or they can simply drop or re-route messages towards other malicious peers. They can magnify their impact by colluding and coordinating their behaviour.

This paper presents PeerCube, a DHT-based system aiming at avoiding high churn from impacting the performance of the system and at the same time at preventing malicious behaviour (coordinated or not) from subverting the system. As many existing DHT-based overlays, PeerCube is based on a hypercubic topology. PeerCube peers self-organise into clusters whose interconnections form the hypercubic topology. Peers within each cluster are classified into two categories, core members and spares, such that only the former ones are actively involved in PeerCube operations. Thus only a fraction of churn affects the overall topology of the hypercube. Defences against eclipse attacks are based on the observation that malicious peers can more easily draw a successful adversarial strategy from a deterministic algorithm than from a randomised one. We show that regardless of the adversarial strategy colluders employ, the randomised insertion algorithm we propose guarantees that the expected number of colluders in each routing table is minimal. Furthermore, by keeping the number of core members per cluster small and constant, it allows to rely on the powerful consensus building block to guarantee consistency of the routing tables despite Byzantine peers. Finally, PeerCube takes advantage of independent and optimal length paths offered by the hypercubic topology to decrease exponentially the probability of encountering a faulty peer with the number of independent paths [22].

To summarise, PeerCube brings together research achievements in both "classical" distributed systems and open large scale systems (Byzantine consensus, clustering, distributed hash tables) so that it efficiently deals with collusion and churn. For the best of our knowledge this work is the first one capable of tolerating collusion by requiring for each `lookup`, `put`, `join` and `leave` operation $\mathcal{O}(logN)$ latency and only $\mathcal{O}(logN)$ messages.

In the remaining of the paper, we discuss related work in Section 2 and then present the system and adversary models in Section 3. Description of the architecture is given in Section 4, together with an analysis of the churn impact. Robustness against malicious behaviours (coordinated or not) is studied in Section 5. Results of simulations are presented in Section 6. We conclude in Section 7.

2 Related Work

In the following, we first review related work that focuses on robustness against malicious peers and then examine policies to handle high churn.

Regarding robustness to malicious behaviour, different approaches have been proposed, each one focusing on a particular adversary strategy. Regarding eclipse attacks, a very common technique, called *constrained routing table*, relies on the uniqueness and impossibility of forging peers' identifiers. It consists in selecting as neighbours only the peers whose identifiers are closer to some particular points in the identifier space [3]. Such an approach has been successfully implemented into several overlays (e.g., CAN, Chord, Pastry). More generally, to prevent messages from being misrouted or dropped, the seminal works on DHT routing security by Castro et al. [3] and Sit and Morris [21] combine routing failure tests and redundant routing as a solution to ensure robust routing. In [16] we extend their approach to cope with colluders by constraining the result of a query, which guarantees to reach the legitimate recipient with high probability. However, in both approaches, the topological properties of their overlay do not guarantee that redundant paths are independent. Fiat et al. [7] use the wide paths technique initially proposed by Hildrum and Kubiatowicz [10]. All these solutions require all DHT nodes to maintain $\mathcal{O}(log^2 N)$ links to other nodes, and require for each operation $\mathcal{O}(log^3 N)$ messages.

With regard to churn, Li and al. [11] show through a comprehensive performance evaluation that structured overlays (such as Tapestry, Chord, or Kademlia) can achieve similar performance with regard to churn if their parameters are sufficiently well tuned. However, these protocols do not focus on reducing the frequency at which routing tables are updated. Such an approach has been proposed in the eQuus architecture [12], in which nodes which are geographically close to each other are grouped into the same cliques to form the vertices of the hypercube. EQuus offers good resilience to churn and good data availability, however relying on local awareness to gather peers within cliques makes this architecture vulnerable to adversarial collusion and geographically correlated failures.

3 Model

3.1 System Model

Peers are assigned unique random identifiers from an m-bit identifier space when they join the system. Identifiers (denoted ID) are derived by using the standard MD5 hash function [17], on the peers' network address. We take the value of m large enough to make the probability of identifers collision negligible. Each application-specific object, or data-item, of the system is assigned a unique identifier, called *key*, selected from the same m-bit identifier space. Each peer p owns a fraction of all the data items of the system. Data items and peers are mapped by a closeness relationship detailed in Section 4.2. In the following, we will use the term *peer* (or *key*) to refer to both the peer (or key) and its m-bit representation. Regarding timing assumption, we assume an asynchronous model. Rational of this assumption is that it matches communication delays over the Internet, and it makes difficult for malicious peers to devise strategies that could have been exploited in a synchronous timing model, such as DoS attacks [14].

3.2 Adversary Model

Some peers try to manipulate the system by not following the prescribed protocols and by exhibiting undesirable behaviours. Such peers are called *malicious*. Malicious peers can drop messages or forward requests to illegitimate peers. Malicious peers may act independently or may be part of a *collusion group*. A peer which always follows the prescribed protocols is said to be *correct*. We assume that there exists a fraction μ, $(0 \leq \mu < 1)$, of malicious peers in the whole system. Malicious peers are controlled by a strong adversary. The adversary can issue join requests for its malicious peers in an arbitrary manner. At any time it can

inspect the whole system and make its malicious peers re-join the system as often as it wishes.

We assume the existence of a public key cryptography scheme that allows each peer to verify the signature of each other peer. We also assume that correct peers never reveal their private keys. Peers IDs and keys are part of their hard coded state, and are acquired via a central authority [5]. When describing the protocols, we ignore the fact that messages are signed and recipients of a message ignore any message that is not signed properly. We also use cryptographic techniques to prevent a malicious peer from observing or unnoticeably modifying a message sent by a correct peer. However a malicious peer has complete control over the messages it sends and receives. Note that messages physically sent between any two correct peers are neither lost nor duplicated.

4 Architecture Description

As discussed before, our architecture is based on a hypercubic topology. The hypercube is a popular interconnection scheme due to its attractive topological properties, namely, low node degree and low network diameter. Beyond these properties, a hypercube offers two important topological features, i.e. recursive construction and independent paths.

4.1 Background

This section presents some preliminaries related to the hypercubic topology. For more details the reader is invited to read Saad and Schultz [19]. A d-dimensional hypercube, or d-hypercube for short, consists of 2^d vertices, where each vertex n is labelled by its d-bits representation. Dimension d is a fundamental parameter since it characterises both the diameter and the degree of a d-hypercube. Two vertices $n = n_0 \dots n_{d-1}$ and $m = m_0 \dots m_{d-1}$ are connected by an edge if they share the same bits but the i^{th} one for some i, $0 \leq i < d$, i.e. if their Hamming distance $\mathcal{H}(n, m)$ is equal to 1. In the following, the notation $n = m^{\overline{i}}$ stands for two vertices n and m whose labels differ only by their bit i.

Property 1 (Recursive Construction [19])**.** *A d-hypercube can be constructed from lower dimensional hypercubes.*

The construction consists in joining each vertex of a $(d-1)$-hypercube to the vertex of the other $(d-1)$-hypercube that is equally labelled, and by suffixing all the labels of the vertices of the first $(d-1)$-hypercube with 0 and those of the second one with 1. The obtained graph is a d-hypercube. From this construction, we can derive a simple distributed algorithm for building a d-hypercube from a $(d-1)$ one. This algorithm consists in splitting each peer of the $(d-1)$-hypercube into two peers, in suffixing the labels of the two obtained peers respectively by 0 and 1, and in updating the link structure to map the d-hypercube. Interestingly, this simple algorithm involves only 2 messages per link updated whatever the dimension of the considered system, and thus has a message complexity of $\mathcal{O}(d)$ per peer.

Property 2 (Independent Routes [19])**.** *Let n and m be any two vertices of a d-hypercube. Then there are d independent paths between n and m, and their length is less than or equal to $\mathcal{H}(n, m) + 2$.*

Two paths are independent if they do not share any common vertex other than the source and the destination vertices. In a d-hypercube, a path from vertex n to vertex m is obtained by crossing successively the vertices whose labels are obtained by modifying one by one n's bits to transform n's label into m's one. Suppose that $\mathcal{H}(n, m) = b$. Then b independent paths between n and m can be found as follows: path i is obtained by successively correcting bit i, bit $i+1$, ..., bit $(i+b-1) \mod b$ among the b different bits between n and m. Note that these b paths are of optimal length $\mathcal{H}(n, m)$. In addition to these paths, $d-b$ paths of length $\mathcal{H}(n, m) + 2$ can be constructed as follows: path j of length $\mathcal{H}(n, m) + 2$ is obtained by modifying first bit j on which n and m agree, and then by correcting the b different bits according to one of the b possibilities described previously, and finally by re-modifying bit

j. The ability to construct independent paths is an important property since it allows the peer-to-peer system to have alternative paths to tolerate faulty peers. Indeed, if the faulty peers are uniformly distributed in the system, then the probability of encountering a faulty peer decreases *exponentially* with the number of independent paths [22]. Finally, finding shortest length paths minimises the probability of encountering malicious peers.

4.2 PeerCube in a Nutshell

We now present an overview of PeerCube features. Basically, our architecture has two main characteristics: peers sharing a common prefix gather together into *clusters*; and clusters self-organise into a hypercubic topology.

4.2.1 Clusters

As stated before, each joining peer is assigned a unique random ID from an m-bit identifier space. Assigning unique random IDs to peers prevents the adversary from controlling a portion of the network, since peers are spread wide over the network according to their identifier as follows. Peers whose ID *share a common prefix* gather together within the same *cluster*. Each cluster is uniquely identified with a *label* that characterises the position of the cluster in the overall hypercubic topology[1]. The label of a cluster is defined as the shortest common prefix shared by all the peers of that cluster such that the *non-inclusion* property is satisfied. The non-inclusion property guarantees that a cluster label never matches the prefix of another cluster label, and thus ensures that each peer in PeerCube belongs to at most one cluster. The non-inclusion property is defined as follows:

Property 3 (Non-Inclusion)**.** *If a cluster $\mathcal{C}$ labelled with $b_0 \dots b_{d-1}$ exists then no cluster $\mathcal{C}'$ with $\mathcal{C}' \neq \mathcal{C}$ whose label is prefixed with $b_0 \dots b_{d-1}$ exists.*

The length of a cluster label, i.e. the number of bits of that label, is called the *dimension* of the cluster. In the following, notation d-cluster denotes a cluster of dimension d. Dimension determines an upper bound on the number of links a cluster has with other clusters of the overlay, i.e. the number of its neighbours. Peers of a d-cluster $\mathcal{C}$ maintain a routing table RT such that entry $RT[i]$, with $0 \leq i < d$, points to peers belonging to one of the d closest clusters to $\mathcal{C}$. (Distance notion is detailed in Section 4.2.2.) References to clusters that point toward $\mathcal{C}$ are maintained by $\mathcal{C}$'s members in a predecessor table PT. Note that maintaining such a data structure is not mandatory, i.e. those clusters can be easily found by the topological properties of PeerCube. However, keeping this information makes the maintenance operations more efficient. Regarding data, all the peers of a cluster are responsible for the same data keys and their associated data. As for most existing overlays, a data key is placed on the closest cluster to this key. Placing a data key on all the peers of a cluster naturally improves fault tolerance since this increases the probability that this key remains available even if some of the peers fail. To keep this probability high, the size of a cluster must not undershoot a certain predefined value S_{min} which depends on the probability of peers' failures. Finally, for scalability reasons, each cluster size is upper bounded by a constant value S_{max} specified later on.

4.2.2 Hypercubic Topology

Clusters self-organise into a hypercubic topology, such that the position of a cluster into the hypercube is determined by its label. Ideally the dimension of each cluster $\mathcal{C}$ should be equal to some value d to conform to a perfect d-hypercube. However, due to churn and random identifier assignment, dimensions may differ from one cluster to another. Indeed, as peers may join and leave the system asynchronously, cluster $\mathcal{C}$ may grow or shrink more rapidly than others. In the meantime, bounds on the size of clusters require that, whenever the size of $\mathcal{C}$ exceeds S_{max}, $\mathcal{C}$ splits into clusters of higher dimensions, and that, whenever the size of $\mathcal{C}$ falls under S_{min}, $\mathcal{C}$ merges with other clusters into a single new cluster of lower dimension.

[1] Henceforth, a cluster will refer to both the cluster and its label.

Finally, since peers IDs, and thus cluster labels, are randomly assigned, some of the labels may initially not be represented at all. For all these reasons dimensions of clusters may not be homogeneous. To keep the structure as close as possible to a perfect hypercube and thus to benefit from its topological properties, we need a *distance* function $\mathcal{D}$ that allows to uniquely characterise the closest cluster of a given label. This is obtained by computing the numerical value of the "exclusive or" (XOR) of cluster labels [13]. To prevent two labels to be at the same distance from a given bit string, labels are suffixed with as many bits "0" as needed to equalise their size to m. This leads to the following distance function $\mathcal{D}$. that makes clusters that share longer prefixes closer to each other.

Definition 1 (Distance $\mathcal{D}$)**.** *Let $\mathcal{C} = a_0 \dots a_{d-1}$ and $\mathcal{C}' = b_0 \dots b_{d'-1}$ be any two d (resp. d') -clusters:*

$$\mathcal{D}(\mathcal{C}, \mathcal{C}') = \mathcal{D}(a_0 \dots a_{d-1} 0^{m-d}, b_0 \dots b_{d'-1} 0^{m-d'}) = \sum_{i=0, a_i \neq b_i}^{m-1} 2^{m-i}$$

$$= \sum_{i=0, a_i \neq b_i}^{m-1} 2^{m-i} \quad (1)$$

Distance $\mathcal{D}$ is such that for any point p and distance Δ there is exactly one point q such that $\mathcal{D}(p, q) = \Delta$ (which does not hold for the Hamming distance). Finally, labels that have longer prefix in common are closer to each other.

We are now ready to detail the content of a cluster's routing table. Let $\mathcal{C} = b_0 \dots b_{d-1}$ and $\mathcal{C}^{\overline{i}} = b_0 \dots \overline{b}_i \dots b_{d-1}$. Then, $\mathcal{C}$'s i^{th} neighbour in PeerCube is cluster $\mathcal{C}'$ whose label is the closest to $\mathcal{C}^{\overline{i}}$.

Property 4. *Let $\mathcal{C}$ be a d-cluster. Then, $\forall i, 0 \leq i < d$, entry i of the routing table of $\mathcal{C}$ is cluster $\mathcal{C}'$ such that for each cluster $\mathcal{C}'' \neq \mathcal{C}'$, $\mathcal{D}(\mathcal{C}^{\overline{i}}, \mathcal{C}') < \mathcal{D}(\mathcal{C}^{\overline{i}}, \mathcal{C}'')$ holds.*

By the distance $\mathcal{D}$ definition, it is easy to see that if for each cluster $\mathcal{C}$ in PeerCube the distance between $\mathcal{C}^{\overline{i}}$ and its i^{th} neighbour is equal to 0 (with $0 \leq i < d$), then PeerCube maps a perfect d-hypercube.

From Property 4, we have:

Lemma 1. *Let $\mathcal{C} = b_0 \dots b_{d-1}$ be a d-cluster. Then for all i, with $0 \leq i < d$, $\mathcal{C}$'s i^{th} neighbour is cluster $\mathcal{C}'$ such that :*

- *$\mathcal{C}'$ is prefixed with $b_0 \dots \overline{b}_i$ if such a cluster exists,*
- *$\mathcal{C}' = \mathcal{C}$ otherwise.*

This can be seen by observing that, by definition of $\mathcal{D}$, $\mathcal{C}'$ shares the longest prefix with $\mathcal{C}^{\overline{i}}$, that is at least the prefix $b_0 \dots \overline{b}_i$. Otherwise $\mathcal{C}$ would be the closest cluster to $\mathcal{C}^{\overline{i}}$. We exploit this property to construct a simple lookup protocol which basically consists in correcting the bits of the source towards the destination from the left to the right.

4.3 Leveraging the Power of Clustering

Dimensions Disparity As described before, clusters dimensions are not necessarily equal to each other. By simply setting $S_{max} > log_2 N$, we can make the dimensions disparity small and constant. Indeed, observe that the dimension of a cluster is necessarily greater than or equal to $log_2 \frac{N}{S_{max}}$. This follows from the fact that the minimum number of clusters is N/S_{max}, which determines the minimum number of bits needed to code the label of a cluster. Furthermore, by setting $S_{max} > log_2 N$, we can show by using Chernoff's bounds that the dimension of a cluster is w.h.p.[2] lower than $log_2 \frac{N}{S_{max}} + 3$. Indeed, since labels are uniformly randomly assigned, setting S_{max} to a higher value decreases clusters dimension.

[2]In the following, with high probability (w.h.p.) means with probability greater than $1 - \frac{1}{N}$.

Thus distance δ between any two clusters dimensions is w.h.p. less than or equal to 3.[3] Furthermore the number of non-represented prefixes is at most 2^3, which is very small with regard to the total number of clusters N/S_{max}. Consequently, by setting $S_{max} > log_2 N$, PeerCube is very close to a $(log_2 \frac{N}{S_{max}})$-hypercube, which guarantees PeerCube to enjoy the attractive topological properties of a perfect hypercube of diameter $log_2 \frac{N}{S_{max}}$. Henceforth S_{max} is in $\Theta(logN)$. Note that more details are given in the appendix.

Limiting the Impact of Churn We have just shown that by having peers self-organised in an hypercube of clusters we get w.h.p. an overlay of diameter $log_2 \frac{N}{S_{max}}$. We now describe how peers take advantage of that clustering to limit the impact of churn on the overall system. Specifically, peers within a cluster are classified into two categories: *core* and *spare* members. Only core members are in charge of PeerCube operations (i.e. inter clusters message forwarding, routing table maintenance, computation of cluster view membership, and keys caching). Size of the core set is equal to the minimal size of a cluster, i.e. constant S_{min}. Core members form a clique, i.e., they point to each other. View of the core set is denoted V_c. In contrast to core members, spare members are temporarily inactive, in the sense that they are not involved in any of the overlay operations. They only maintain links to a subset of core members of their cluster and cache the set of keys and associated data as core members do. Within a cluster, apart from the core members that maintain the view V_s of the spares set, no other peer in the system is aware of the presence of a particular spare, not even the other spares of the cluster. As a consequence, routing tables only point to core members, that is S_{min} references per entry are needed.

Achieving High Consistency By keeping the size of the core set to a small and constant value, we can afford to rely on the powerful consensus building block to guarantee consistent routing tables among correct core members despite the presence of a fraction μ of Byzantine peers among them. Briefly, in the consensus problem, each process proposes a value, and all the non-faulty processes have to eventually decide (termination property) on the same output value (agreement property), this value having been proposed by at least one process (validity property). Various Byzantine consensus algorithms have been proposed in the literature (good surveys can be found in [8, 4]). In PeerCube, we use the solution proposed by Correia et al. [4] essentially because it provides optimal resiliency, i.e. tolerates up to $\frac{n-1}{3}$ Byzantine processes in a group of n processes, and guarantees that a value proposed only by Byzantine processes is never decided by correct ones. Message complexity is in $\mathcal{O}(n^3)$. Note that in our context, $n = S_{min} = \text{cst}$.

4.4 PeerCube Operations

From the application point of view, three key operations are provided by the system: the `lookup`(k) operation which enables to search for key k, the `join` operation that enables a peer to join the system, and the `leave` operation, indicating that some peer left the system. Note that the `put`(x) operation, that enables to insert data x in the system, is not described since it is very similar to the `lookup`() operation. From the topology structure point of view, three events may result in a topology modification: when the size of a cluster exceeds S_{max}, this cluster *splits* into two new clusters; when the size of a cluster reaches S_{min}, this cluster *merges* with other clusters to guarantee the cluster resiliency; finally, when a peer cannot join any existing cluster because none of them matches the peer identifier prefix, then a new cluster is *created*. For robustness reasons, a cluster may have to temporarily exceed its maximal size S_{max} before being able to split into two new clusters. This guarantees that resiliency of both new clusters is met, i.e both clusters sizes are at least equal to S_{min}. A similar argument applies to the `create` operation. For this specific operation, peers whose identifiers do not match any cluster label, temporarily join the closest cluster to their identifier, and whenever $T_{split} \geq S_{min}$ temporary peers share the same prefix then they create their new cluster. Threshold T_{split} is discussed in Section 4.4.2. These three additional

[3]Note that for a pure hypercube, the dimension disparity is $log_2 N$.

```
Upon lookup(k) from the application do
  if (p.type ≠ {core}) then
    {q_0 ... q_⌊(S_min−1)/3⌋} ← p.coreRandomPeer();
    p sends (LOOKUP, k,p) to {q_0 ... q_⌊(S_min−1)/3⌋}
  else
    C ← p.findClosestCluster(k);
    p sends (LOOKUP,k,p) to a random subset of
    ⌊(S_min − 1)/3⌋ + 1 peers in C.coreSet;
enddo
Upon receiving (LOOKUP,k,q) from the network do
  C ← p.findClosestCluster(k);
  if (p.cluster.label= C) then
    p sends (LOOKUP, k,q) to core members in C
    if not already done;
    data ← k's data if cached otherwise null;
    sends (k,C,data) to the originating q by using the reverse path;
  else
    p sends (LOOKUP, k,q) to a random subset of
    ⌊(S_min − 1)/3⌋ + 1 peers in C.coreSet;
enddo
findClosestCluster(k)
  if (p.dim=0 or p.cluster.prefix(k)) then
    C ← p.cluster;
  else
    C.label ← RT_p(0).label;
    for (i = 0 to p.dim − 1) do
      if (D(k,RT_p(i).label) < (D(k,C.label))) then
        C.label ← RT_p(i).label;
  return C;
```

Figure 1: `lookup` Operation at Peer p

operations exploit the recursive construction property of hypercubes to minimise topology changes, and rely on the Byzantine-consensus building block to achieve high consistency among routing tables.

4.4.1 `lookup` Operation

In this section we describe how peer $p \in \mathcal{C}$ locates a given key k through the `lookup` operation. Basically, locating k consists in walking in the overlay by correcting one by one and from left to right the bits of p's identifier to match k. By Lemma 1 and by distance $\mathcal{D}$, this simply consists in recursively contacting the closest cluster to k. In a failure free environment, this operation would be similar to a typical `lookup` operation, except that if the originator p of the `lookup` was a spare member, then p would forward its request to a randomly chosen core member of $\mathcal{C}$. Then the request would be propagated until finding either a peer of a cluster labeled with a prefix of k, or no cluster closer to k than the current one. The last contacted peer would return to the originating peer p either the requested data if it exists, or null otherwise.

Now, suppose that malicious peers may drop or misroute requests they receive to prevent them from reaching their legitimate destination. We adapt the `lookup` operation by using the *width path* approach, commonly used in fault tolerant algorithms, which consists in forwarding a request to sufficiently enough peers so that at least one correct peer receives it. This is described in Figure 1. Specifically, a request is forwarded to $\lfloor (S_{min} - 1)/3 \rfloor + 1$ randomly chosen core members of the closest cluster to the request destination, instead of only one randomly chosen core member as in the basic `lookup` operation. In addition, in the last contacted cluster $\mathcal{C}$, when a core member $p \in \mathcal{C}$ receives the request, if p has not already sent it to all core members of $\mathcal{C}$ then it does so and returns the response through the reverse path. Hence, each peer that forwarded the request waits for a quorum of responses (i.e., $\lfloor (S_{min} - 1)/3 \rfloor + 1$) before propagating the response back in the reverse path. When the originator q of the `lookup` request receives $\lfloor (S_{min} - 1)/3 \rfloor + 1$ similar responses (k,*data*,$\mathcal{C}$)

issued from peers whose ID prefix matches the one q initially contacted, then q can safely use the received *data*. Otherwise, q discards it. It is easy to see that if there are no more than $\lfloor (S_{min} - 1)/3 \rfloor$ malicious core members per cluster crossed, then a lookup operation invoked by a correct peer returns the legitimate response.

Lemma 2. *The* `lookup`(k) *operation returns the data associated to* k *if it exists, null otherwise. This is achieved in* $\mathcal{O}(logN)$ *hops and requires* $\mathcal{O}(logN)$ *messages.*

Proof. Let us first suppose a failure free-environment. Consider a peer p invoking `lookup`(k). First, observe that since at each hop, the next contacted cluster is necessarily closer to k than the current one, the lookup eventually terminates.

Let $\mathcal{N}'$ be the cluster of the core member returned by the `lookup`(k) operation. To prove that $\mathcal{N}'$ is the closest cluster to k we proceed by contradiction. Suppose that there exists some cluster $\mathcal{N}''$ such that $\mathcal{D}(\mathcal{N}'', k) < \mathcal{D}(\mathcal{N}', k)$. By definition of $\mathcal{D}$, if we denote the position of the leftmost different bit between $\mathcal{N}'$ and $\mathcal{N}''$ by i, it must be the case that $\mathcal{N}''$ and k share their i^{th} bit:

- $\mathcal{N}' = a_0 \dots a_i a_{i+1} \dots$
- $\mathcal{N}'' = a_0 \dots \overline{a}_i b_{i+1} \dots$
- $k = a_0 \dots \overline{a}_i c_{i+1} \dots$

By Property 3, i necessarily exists. Then since a cluster prefixed with $a_0 \dots \overline{a}_i$ exists ($\mathcal{N}''$), by Lemma 1 the i^{th} neighbour $\mathcal{N}'_i$ of $\mathcal{N}'$ is prefixed with $a_0 \dots \overline{a}_i$, that is $\mathcal{N}'_i$ necessarily shares its i^{th} bit with k.

- $\mathcal{N}'_i = a_0 \dots \overline{a}_i a'_{i+1} \dots$

Thus $\mathcal{N}'_i$ is closer to k than $\mathcal{N}'$ is. But by the algorithm, $\mathcal{N}'$ should have forwarded the request for k to $\mathcal{N}'_i$. This contradicts the assumption and thus proves the first part of the lemma.

Let us now prove that the lookup stops after $O(logN)$ hops. First, since the distance to k stricly decreases at each hop, at least one bit is corrected. Second, from the distance definition and by Lemma 1, bits are corrected from the left to the right, and a corrected bit is never flipped again. Then the maximum number of hops is not greater than twice the maximum dimension of a cluster (twice because of the reverse path). Since the dimension of a cluster is w.h.p. at most $log_2 N - log_2 S_{max} + 3$ (see Lemma 17), the number of hops needed to complete a lookup is w.h.p. in $O(logN)$. Trivially, this operation requires $O(logN)$ messages.

In a failure prone environment, message redundancy is added. This redundancy impacts only the number of messages required to complete the operation. Not its latency. By construction, the request is no more forwarded to one peer at each hop, but to $\lfloor (S_{min} - 1)/3 \rfloor + 1$ peers. Since . Thus since S_{min} is constant, this does not influence the failure free operation asymptotic complexity. This completes the proof of the lemma. □

4.4.2 `join` Operation

Recall that by construction each cluster $\mathcal{C}$ contains all the core and spare members p such that $\mathcal{C}$'s label is a prefix of p's ID, and that each peer p belongs to a unique cluster. To join the system, peer p sends a `join` request to a correct peer it knows in the system. The request is forwarded until finding the closest cluster $\mathcal{C}$ to p's ID. Two cases are possible: either $\mathcal{C}$'s label matches the prefix of p's ID or the cluster $\mathcal{N}$ p should be inserted into does not already exist ($\mathcal{C}$ is only the closest cluster to $\mathcal{N}$). In the former case, p is inserted into $\mathcal{C}$ as a spare member. Inserting newcomers as spare members prevent malicious peers from designing deterministic strategies to increase their probability to act as core member. In the latter case, p is temporarily inserted into $\mathcal{C}$ until creation of $\mathcal{N}$ is possible, i.e., predicate `tempIsSplit()` in Figure 2 holds. This predicate holds if there exist T_{split} temporary peers

```
Upon join(p) from the application do
  {q_0 ... q_⌊(S_min−1)/3⌋} ← findBootstrap();
  p sends (JOIN,p) to q ∈ {q_0 ... q_⌊(S_min−1)/3⌋};
enddo;
Upon receiving (JOIN,q) from the network do;
  C ← p.findClosestCluster(q.id);
  if (p.cluster= C) then
    if (p.cluster.prefix(q.id)) then
      p broadcasts (JOINSPARE,C,q) to p's core set;
    else
      p broadcasts (JOINSTEMP,C,q) to p's core set;
  else
    p sends (JOIN,q) to a random subset of
    ⌊(S_min − 1)/3⌋ + 1 peers in C's core set;
enddo;
Upon delivering (JOINSPARE,C,q) from the network do;
/* each core member ∈ C executes this code once by the
broadcast properties */
  V_s ← V_s ∪ q;
  if (p.clusterIsSplit) then p.split();
  N=p.findClosestCluster(q.id);
  p sends (JOINACK,N,state) to q;
enddo;
Upon delivering (JOINSTEMP,C,q) from the network do;
/* each core member ∈ C executes this code once by the
broadcast properties */
  p.temp← p.temp ∪ q;
  if (p.tempIsSplit) then p.create(p.temp);
  C'=p.findClosestCluster(q.id);
  p sends (JOINACK,N,state) to q;
enddo;
```

Figure 2: `join` Operation at Peer p

in $\mathcal{C}$ that share a common prefix. Note that temporary peers do not participate in the cluster life (they do not even cache data, contrary to spares), and only core members are aware of their presence. Threshold T_{split} is introduced to prevent the adversary from triggering a "split-merge" cyclic phenomenon. Indeed, a strong adversary can inspect the system and locate the clusters that are small enough so that the departure of malicious peers from that cluster triggers a merge operation with other clusters, and their re-joining activates a split operation of the newly created cluster. Thus by setting $T_{split} - S_{min} > \lfloor \frac{S_{max}-1}{3} \rfloor$ with $\lfloor \frac{S_{max}-1}{3} \rfloor$ the expected number of malicious peers in a cluster, probability of this phenomenon is negligible. In both cases, i.e. whether p is inserted as spare or temporary peer of $\mathcal{C}$, p's insertion is broadcast to all core members. The *broadcast* primitive guarantees that if a correct sender broadcasts some message m, then all correct recipients eventually deliver m once[4]. Peer p's insertion in a cluster is acknowledged to p by all correct core members of p's new cluster via a JOINACK message which carries information (*state*) that p needs to join its cluster (whether p is spare or temporary, and the required data structures, if any). In all cases, a constant number of messages are needed. Thus message complexity of a `join` is $\mathcal{O}(logN)$ which is the cost of the lookup for $\mathcal{C}$.

Lemma 3. *The `join` operation is insensitive to collusion. That is if before a join operation in $\mathcal{C}$ the expected number of malicious peers in $\mathcal{C}$ is $\mu.S_{min}$, then after a join in $\mathcal{C}$ the expected number of malicious peers is still equal $\mu.S_{min}$.*

Proof. Straightforward from the join algorithm. □

[4]PeerCube relies on the asynchronous Byzantine-resistant reliable broadcast of Bracha [2], whose time complexity is in $\mathcal{O}(1)$ (the protocol runs in exactly 3 asynchronous rounds, where an asynchronous round involves the sender sending a message and receiving one or more messages sent by recipients), and message complexity is in $\mathcal{O}(n^2)$. As for consensus, it is important to note that in our case $n = S_{min} = \text{cst}$.

```
leave(p) /* run by core member p upon q's departure*/
Upon (q's failure detection) do
  if (q ∈ V_s) then V_s ← V_s \ {q};
  else
    p chooses S_min random peers R = {r_1, ..., r_j} in V_s ∪ V_c;
    {s_1, ..., s_j} ← run consensus on R among V_c members;
    /* the decision value is delivered at all core members */
    p.leavePredTable() ;
    V_s ← V_s ∪ V_c \ {s_1, ..., s_j};
    V_c ← {s_1, ..., s_min};
    p sends (LEAVE, V_c) to all spare members ∈ V_s;
    p.leaveRoutingTable();
enddo;
```

Figure 3: `leave` Operation at Peer p

4.4.3 `leave` Operation

The `leave` operation is executed when a peer q wishes to leave a cluster or when q's failure has been detected. Note that in both cases, q's departure has to be detected by $\lfloor (2S_{min} + 1)/3 \rfloor + 1$ core members so that a malicious peer cannot abusively pretend that some peer q left the system. Thus, when core members detect that q left, two scenarios are possible. Either q belonged to the spare set, in which case, core members simply update their spare view to reflect q's departure, or q belonged to the core set. In the latter case, q's departure has to be immediately followed by the core view maintenance to ensure its resiliency (and thus the cluster resiliency). To prevent the adversary from devising collusive scenario to pollute the core set, the whole composition of the core set has to be refreshed. Indeed, replacing the peer that left by a single one (even randomly chosen within the spare set) does not prevent the adversary from ineluctably corrupting the core set: once malicious peers succeed in joining the core set, they maximise the benefit of their insertion by staying in place; this way, core sets are eventually populated by more than $\lfloor \frac{S_{min}-1}{3} \rfloor$ malicious peers, and thus become – and remain – corrupted. This is illustrated in Section 5. Thus each core member chooses S_{min} random peers among both core and spare members, and proposes this subset to the consensus. By the consensus properties, a single decision is delivered to all core members, and this decision has been proposed by at least one correct core member. Thus core members agree on a unique subset which becomes the new core set. Note that in addition to preventing collusion, refreshing the whole core set guarantees that the expected number of malicious peers in core sets, and thus the number of corrupted entries in routing tables is bounded by μS_{min} which is minimal:

Lemma 4. *After a core member's departure, the expected number of malicious peers in that core is at most μS_{min}.*

Proof. Consider a cluster $\mathcal{C}$. Since IDs are uniformly randomly chosen, the expected fraction of malicious peers in $\mathcal{C}$ is at most μ. When a core member leaves the cluster, S_{min} peers are chosen randomly among all the peers of $\mathcal{C}$. Thus, the expected number of malicious peers included in the new core is at most μS_{min}. □

An adversarial strategy is determined by the arrival or departure of a malicious peer at a given time step.

Lemma 5. *Upon a core member's departure, for any randomized algorithm, there exists an adversarial strategy such that the expected number of malicious peers in the core is at least μS_{min}.*

Proof. By Yao's lemma [24], it is sufficient to prove that the lower bound holds for any deterministic algorithm, on average, for some probability distribution over the adversarial strategies (join/leave). Let us assume that the probability that a malicious peer joins (resp. leaves) $\mathcal{C}$ is equal to the probability that a correct peer joins (resp. leaves) $\mathcal{C}$. From

```
split(C) /* run by core member p in C*/
  label_0, label_1 ← the two shortest non common prefixes
  shared by at least T_split peers in C
  for (i=0,1) do
    V_{c_i} ← {q | q ∈ C's core set and q's prefix is label_i};
    V_{s_i} ← {q | q ∈ C's spare set and q's prefix is label_i};
    p chooses V_i = S_min − | V_{c_i} | random peers from V_{s_i}
  enddo;
    (V_0', V_1') ← run consensus on (V_0, V_1) among V_c members;
    if (p ∈ V_0') then i = 0 else i = 1;
    p updates its cluster label and dimension;
    V_{c_i} ← V_{c_i} ∪ V_i'; V_{s_i} ← V_{s_i} \ V_i';
    p.splitRoutingTable() to update RT;
    p.splitPredTable() to update PT;
    p sends (SPLIT,label'_i,state) to all peers in V_{s_i};
```

Figure 4: `split` Operation at Peer p

Lemma 4.4.2 the expected number of malicious peers in the core is μS_{min}. Consider a deterministic algorithm that minimises the number of malicious peers in the core after a core member p leaves. Such an algorithm would simply choose an arbitrary peer q in the spares set to replace p (e.g. the one with the lowest ID as an arbitrary rule). Then, since malicious peers join and leave $\mathcal{C}$ at the same rate as correct peers, the expected probability that the leaving peer p is malicious is $\mu S_{min}/S_{min} = \mu$. However, the expected probability that peer q replacing p is malicious is μ. Hence, the expected number of malicious peers in the core remains equal to μS_{min}. □

Remark that because of the asynchrony of the system, some of the agreed peers s_i may still belong to some views while having been detected as failed or left by others, or may belong to only some views because of their recent join. In the former case, all the correct core members eventually deliver the consensus decision notifying s_i's departure, and new consensus is run to replace it. Note that for efficiency reason, each core member can ping the peers it proposes before invoking the consensus. In the latter case, s_i's recent arrival is eventually notified at all correct core members by properties of the broadcast primitive (see `join` operation), and thus they insert s_i in V_s. Then each core member p notifies all the clusters that point to $\mathcal{C}$ (i.e. entries of p's PT table) of $\mathcal{C}$'s new core set. Core members of each such cluster can safely update their entries upon receipt of $\lfloor \frac{S_{min}-1}{3} \rfloor + 1$ similar notifications. This is encapsulated into the `leavePredTable()` procedure in Figure 3. Similarly, all the peers $\{s_1, \ldots, s_{min}\}$ are safely notified about their new state, and locally handle the received data structures (invocation of `leaveRoutingTable()` procedure). Former core members only keep their keys and the associated data. In all cases a constant number of messages are exchanged for a `leave`.

4.4.4 `split` Operation

As discussed above, when the size of a cluster $\mathcal{C}$ exceeds the S_{max} threshold, then $\mathcal{C}$ has to split into two new clusters. We exploit the recursive construction property of hypercubes to achieve a `split` operation involving $\mathcal{O}(logN)$ messages.

When locally some core member p of a d-cluster $\mathcal{C}$, with $1 \leq d \leq m$, detects that the conditions to split its cluster are satisfied (i.e., predicate `clusterIsSplit` holds) then p invokes the `split` operation. Let $a_0 \ldots a_{d'-1}$ and $a_0 \ldots \overline{a}_{d'-1}$, with $d' > d$ be the *shortest non-common prefixes* shared by $\mathcal{C}$'s members identifiers (i.e., core and spare members).

The `split` operation is described in Figure 4. The first step consists in building the core sets of the two new clusters $\mathcal{C}'$ and $\mathcal{C}''$. Specifically, new core sets are prioritarily populated with core members of $\mathcal{C}$ and then completed with randomly chosen spares of $\mathcal{C}$. This is handled as in the `leave` operation through consensus invocation by core members. The second step consists, for each core member $p \in \mathcal{C}'$ (similarly for $p \in \mathcal{C}''$), in updating its views V_c and V_s by removing all the peers that do not anymore share a common prefix with

its new cluster, and similarly with data and associated keys (p keeps data closest to its new cluster). Peer p sends to V_s members the label of their new cluster and the new core view. Peers in V_s then remove any keys and associated data that do not belong to the cluster anymore. The final step consists in updating routing tables of $\mathcal{C}'$ and $\mathcal{C}''$, as well as the ones that pointed to $\mathcal{C}$ prior to its splitting (i.e. entries in the predecessor table) as follows.

We adapt the distributed algorithm sketched in Section 4.1 to a imperfect hypercube as follows: This procedure is referred as `splitRoutingTable()` in Figure 4. Consider w.l.o.g cluster $\mathcal{C}' = a_0 \dots a_{d'-1}$ (the same argument applies for $\mathcal{C}''$). By construction of routing tables, the $(d'-1)^{th}$ neighbour of $\mathcal{C}'$ is set to $\mathcal{C}''$. Now, for all $i = 0 \dots d-1$, if the dimension of the i^{th} neighbour of $\mathcal{C}$ was greater than d, then the i^{th} neighbour of $\mathcal{C}'$ is found by invoking `lookup`($\mathcal{C}'^{\overline{i}}$). Otherwise, the i^{th} neighbour of $\mathcal{C}'$ remains equal to the i^{th} neighbour of $\mathcal{C}$. Finally, for all j, such that $d-1 < j < d'-1$, the j^{th} neighbour of $\mathcal{C}'$ is set to $\mathcal{C}'$ itself.

We now describe how routing tables of clusters that used to point to $\mathcal{C}$ are updated to reflect the creation of the two new clusters $\mathcal{C}'$ and $\mathcal{C}''$. Let Γ be the set of these clusters, and $\mid \Gamma \mid = k$. Each core member p (at least each correct one) that used to be in $\mathcal{C}$ contacts all $q \in Pred_{\mathcal{C}}[i].core$, with $0 \leq i < k$. Then for each d''-cluster $\mathcal{Q} = b_0 \dots b_{d''-1} \in \Gamma$, let $i \leq d''-1$ be the entry that referred to $\mathcal{C}$. If $\mathcal{Q}^{\overline{i}}$ is closer to $\mathcal{C}'$ than it is to $\mathcal{C}''$, then it is set to $\mathcal{C}'$, otherwise it is set to $\mathcal{C}''$. The routing tables of clusters that did not refer to $\mathcal{C}$ are left unchanged. This procedure is referred as `splitPredTable()` in Figure 4. Note that the `split` operation has been described by assuming the creation of two clusters. This can be easily extended to $k \geq 2$ clusters. This completes the `split` operation.

Lemma 6. *Let $\mathcal{C}$ be some d-cluster that splits into two d'-clusters $\mathcal{C}'$ and $\mathcal{C}''$, with $d \leq d' \leq m$. If $\mathcal{C}$ satisfied Properties 3 and 4 prior to splitting, then both $\mathcal{C}'$ and $\mathcal{C}''$ satisfy Properties 3 and 4.*

Proof. We prove first that Property 3 holds. By assumption $\mathcal{C}$ initially satisfies Property 3. Thus there is no cluster prefixed with $\mathcal{C}$. Then after the invocation of `split`, $\mathcal{C}'$ and $\mathcal{C}''$ are the only two clusters prefixed with $\mathcal{C}$. From the `split` algorithm, $\mathcal{C}'$ and $\mathcal{C}''$ differ by their last bit. Thus neither $\mathcal{C}'$ nor $\mathcal{C}''$ is prefixed with the other one. Thus $\mathcal{C}'$ and $\mathcal{C}''$ satisfy Property 3.

We now prove that Property 4 holds. Let us focus on cluster $\mathcal{C}'$ (case for $\mathcal{C}''$ is similar). Let $\mathcal{C} = b_0 \dots b_{d-1}$, $\mathcal{C}' = b_0 \dots b_{d'-1}$, and $\mathcal{C}'' = b_0 \dots \overline{b}_{d'-1}$. Observe first that as $\mathcal{C}'$ and $\mathcal{C}''$ differ by exactly their $(d'-1)^{th}$ bit, $\mathcal{C}''$ is trivially the $(d'-1)^{th}$ neighbour of $\mathcal{C}'$. Consider now the j^{th} neighbor of $\mathcal{C}'$ such that $d-1 < j < d'-1$. Since no cluster other than $\mathcal{C}'$ and $\mathcal{C}''$ prefixed with $\mathcal{C}$ exists, no cluster prefixed with $b_0 \dots b_{d-1} \dots \overline{b}_{j-1}$ exists. Then by Lemma 1, the j^{th} neighbor of $\mathcal{C}'$ has to be set to $\mathcal{C}'$ itself. Finally, consider the j^{th} neighbor of $\mathcal{C}'$ such that $0 \leq i \leq d-1$. Two cases have to be considered. First, the dimension of that neighbour cluster is smaller than or equal to d. By construction, the prefix of length d of $\mathcal{C}'$ is $\mathcal{C}$. By the distance definition, that j^{th} neighbour remains the closest cluster to $\mathcal{C}'^{\overline{i}}$. In the second case, by Theorem 2 the i^{th} neighbour of $\mathcal{C}'$ found by invoking `lookup`($\mathcal{C}'^{\overline{i}}$) is the closest cluster to $\mathcal{C}'^{\overline{i}}$. □

Lemma 7. *Let $\mathcal{C}$ be some d-cluster that splits into two d'-clusters $\mathcal{C}'$ and $\mathcal{C}''$, with $d \leq d' \leq m$. Let $\mathcal{O}$ be any d'-cluster different from both $\mathcal{C}'$ and $\mathcal{C}''$. Then if $\mathcal{O}$ satisfied Properties 3 and 4 prior to the splitting operation, then $\mathcal{O}$ satisfies Properties 3 and 4 after that split.*

Proof. An argument similar to the one used in Lemma 6 shows that Property 4 holds. Let $\mathcal{O}$ be some d'-cluster, with $1 \leq d' \leq m$. Two cases are possible:

- $\mathcal{O}$ did not have $\mathcal{C}$ as neighbour. We show that $\mathcal{O}$ is not affected by $\mathcal{C}$'s split. Let $\mathcal{N}_i$ be some i^{th} neighbour of $\mathcal{O}$, with $0 \leq i \leq d'-1$. Then by construction $D(\mathcal{O}^{\overline{i}}, \mathcal{N}_i) < D(\mathcal{O}^{\overline{i}}, \mathcal{C})$. Since $\mathcal{C}'$ is prefixed with $\mathcal{C}$, by the definition of $\mathcal{D}$, $\mathcal{D}(\mathcal{O}^{\overline{i}}, \mathcal{N}_i) < \mathcal{D}(\mathcal{O}^{\overline{i}}, \mathcal{C}')$. The proof is similar for $\mathcal{C}''$. This shows that $\mathcal{O}$ is not affected by $\mathcal{C}$'s split.

- $\mathcal{O}$ had $\mathcal{C}$ as neighbour, that is, there exists some i, such that the i^{th} neighbour of $\mathcal{O}$ was $\mathcal{C}$. By construction, $\mathcal{C}$ was the closest cluster to $\mathcal{O}^{\overline{i}}$. By the split algorithm, only

$\mathcal{C}'$ and $\mathcal{C}''$ are prefixed with $\mathcal{C}$, and one of them is the new i^{th} neighbour of $\mathcal{O}$. Thus, by the routing table update algorithm, Property 4 holds.

□

Lemma 8. *The number of messages involved in the* `split` *operation is in* $\Theta(logN)$.

Proof. Recall that a `split` proceeds in two steps: first a split message is sent to the peers of the splitting cluster, second routing tables of the created clusters as well as the ones of clusters that should point to them are updated. In the first step, since the size of a cluster is in $\Theta(logN)$, the number of messages exchanged is obviously $\Theta(logN)$. In the second step, two substeps are executed to update routing tables.

Consider a core member p of a splitting cluster $\mathcal{C} = b_0 \dots b_{d-1}$. Denote by $\mathcal{C}' = b_0 \dots b_{d-1} \dots b_{d'-1}$ the label of $\mathcal{C}'$ after the split is completed and suppose that $p \in \mathcal{C}'$. First, the i^{th} entry of the routing table of p is updated only when the dimension of the cluster $\mathcal{C}_i$ initially referred by entry i of $\mathcal{C}$ is greater than the initial dimension d of $\mathcal{C}$ (see Section 4.4.4). In this case, p executes `lookup`($\mathcal{C}'^{\overline{i}}$) and updates the entry to refer to the found cluster. Actually, the `lookup` consists in correcting only the bits b_d to $b_{d'-1}$ of $\mathcal{C}_i$ up to the `split`. Indeed, since $\mathcal{C}_i$ is already the closest cluster to $\mathcal{C}^{\overline{i}}$, only the $d' - d$ last bits of $\mathcal{C}_i$ need to be corrected. But since w.h.p. dimensions d and d' differ by only a constant number, the `lookup` operation involves only a constant number of messages. Thus, since only core peers handle routing tables and since the number of core peers is bounded by a constant number, the number of messages exchanged when updating the entries of the routing table is $O(logN)$.

Second, p contacts the clusters in Γ initially pointing to $\mathcal{C}$ through `pred`. Since the cost of `pred` is $O(logN)$ and since only a constant number of peers per cluster (core members) are involved in the procedure, this operation consumes $O(logN)$ messages.

Thus the number of messages involved in a `split` is $\Theta(logN)$. □

Lemma 9. *After a SPLIT operation, the expected number of malicious peers in each core is at most* μS_{min}.

Proof. Consider a cluster $\mathcal{C}$. Since identifiers are uniformly randomly chosen, the expected fraction of malicious peers in $\mathcal{C}$ is at most μ. When $\mathcal{C}$ splits, since identifiers are randomly assigned, the expected fraction of malicious peers in each core is μ. Similarly, the expected fraction of malicious peers in each spares set is μ. Since the new cores are filled with randomly chosen spares, the expected fraction of malicious peers in each core remains equal to μ. Thus, the expected number of malicious peers in each core is μS_{min}. □

Lemma 10. *Upon a SPLIT operation, for any randomized algorithm, there exists an adversarial strategy such that the expected number of malicious peers in the core is at least* μS_{min}.

Proof. By Yao's lemma [24], it is sufficient to prove that the lower bound holds for any deterministic algorithm, on average, for some probability distribution over the adversarial strategies (join/leave). Let us assume that the probability that a malicious peer joins (resp. leaves) $\mathcal{C}$ is equal to the probability that a correct peer joins (resp. leaves) $\mathcal{C}$. Assume that the expected number of malicious peers in the core prior to splitting is μS_{min}. Note first that, after the SPLIT operation, since ids are randomly assigned, the expected fraction of malicious peers in each core is μ. Consider a deterministic algorithm that minimises the number of malicious peers in each core after a SPLIT operation. Such an algorithm would simply choose arbitrary peers in the spares set to fill the core (e.g. the ones with the lowest ids). Then, since malicious peers join and leave $\mathcal{C}$ at the same rate as correct peers, the expected fraction of malicious peers among those chosen spares is μ. Hence, the expected fraction of malicious peers in each core remains equal to μ. □

4.4.5 `merge` Operation

We now describe how PeerCube is updated when the size of a cluster falls under S_{min}. The `merge` operation is the dual to the `split` one, and incurs the same cost in terms of number of messages exchanged. The `merge` consists in gathering the cluster that has detected the need to merge with all the clusters that share the longest common prefix with that cluster. Specifically, suppose that core member $p \in \mathcal{C} = b_0 \dots b_{d'-1}$, locally detects that its cluster size falls under S_{min}. First, p searches the set of clusters that share as prefix the bit string $b_0 \dots \overline{b}_{d'-1}$. This set, named Γ, is returned by invocation of the function `commonPrefix`$(b_0 \dots \overline{b}_{d'-1})$. This search function is based on a constrained flooding approach. The constraint prevents a cluster from being contacted twice. It is directly derived from the property of routing tables (Lemma 1). This makes this function optimal in the number of messages exchanged, i.e. equal to the number of clusters prefixed by the bit string. From the dimension disparity remark, the number of clusters that share as prefix the bit string is constant. Thus a constant number of messages are sent during the search. Secondly, p broadcasts to the other core members of $\mathcal{C}$ the fact that their cluster has to merge with the clusters in Γ.

All the clusters $\mathcal{Q} \in \Gamma$ merge with $\mathcal{C}$ as follows. First, the core set of the cluster with the lowest label in $\mathcal{C} \cup \Gamma$ is kept as the core set of the new cluster, and spare members from this cluster together with all the core and spare members of the other merging clusters are included in the spare set of the new cluster. Both core and spare members update their keys by copying those from the other merging clusters. Temporary peers belonging to all the merging clusters keep their temporary status in the new cluster. Dimension d of the new cluster is the lowest dimension so that Property 3 holds. Finding that dimension comes to choose the value of the largest entry of $\mathcal{C}$ that does not point to itself. In most cases, $d = d' - 1$. The new cluster label $\mathcal{N}$ is $\mathcal{N} = b_0 \dots b_{d-1}$. Second, routing tables of

```
merge(C) /* run by core member p in the d'-cluster C*/
  Γ ← p.commonPrefix(C);
  p broadcasts (MERGE, Γ) to core members of C;
  Upon delivery (MERGE,Γ) do
    N.dim ← i | RT[i] ≠ C ∧ (RT[i+1] = C ∨ i ≥ d');
    N.label ← the leftmost first d bit of label of C;
    V_c^N ← V_c^C; V_s^N ← V_s^C; N.temp ← C.temp;
    for (each Q ∈ Γ) do
      V_s^N ← V_s^N ∪ V_c^Q ∪ V_s^Q; N.temp ← N.temp ∪ Q.temp;
      p.mergeIncomingLinks();
      p.mergeOutgoingLinks();
    enddo
    p sends (MERGRq, N, state) to each q ∈ V_s^N ∪ temp;
  enddo
```

Figure 5: `merge` Operation at Peer p

clusters that used to point toward the merging clusters $\mathcal{C} \cup \Gamma$ are updated to reflect the merge operation. Core members of each cluster $\mathcal{Q} \in \Gamma$ notify their predecessors (i.e., entries of their predecessor table) to update the entry that used to point to $\mathcal{Q}$ so that it now points to peers in the core set of the new cluster $\mathcal{N}$. Note that regarding clusters that used to point toward $\mathcal{C}$, only label of $\mathcal{C}$'s label has to be replaced by $\mathcal{N}$'s one. This function is referred as `mergeIncomingLinks()` in Figure 5. Finally, $\mathcal{N}$'s routing table has to be created. Specifically, if bit $(d'-1)$ of $\mathcal{C}$ label is bit "0" then $\mathcal{N}$'s routing table entries are set to entries $0 \dots (d-1)$ of $\mathcal{C}$'s routing table. Otherwise $\mathcal{N}$'s routing table entries are set with the $(d-1)$ first entries of the $(d'-1)^{th}$ neighbour of $\mathcal{C}$. In both cases entries $d \dots (d'-1)$ do not exist anymore. This function is referred as `mergeOutgoingLinks()` in Figure 5. This completes the `merge` operation.

Lemma 11. *Let $\mathcal{C}$ be some d'-cluster that merges with all the clusters of Γ into a unique d-cluster $\mathcal{C}$, with $d < d' \leq m$. Then, if all clusters satisfied Properties 3 and 4 prior to the merge operation, then they all satisfy Properties 3 and 4 after the merge.*

Proof. First note that, trivially, for all clusters $\mathcal{O}$ different from $\mathcal{C}'$ and not in Γ, if $\mathcal{O}$ satisfies Property 3 before the merge operation, then it satisfies it after the merge.
Let us now prove that $\mathcal{C}$ satisfies Property 3. First, since $\mathcal{C}'$ satisfies Property 3, no cluster prefixed with $\mathcal{C}'$ exists. From the algorithm, all clusters prefixed with $b_0 \dots \overline{b}_{d'-1}$ and $\mathcal{C}'$ exists are merged in $\mathcal{C}$. From the algorithm, d is the highest value such that the $(d-1)^{th}$ neighbor of $\mathcal{C}'$ is not $\mathcal{C}'$ itself. By Lemma 1, no cluster prefixed with $b_0 \dots b_{d-1} \dots \overline{b}_{i'-1}$ for all $d+1 \leq i' \leq d'-1$ exists. Therefore no cluster prefixed with $\mathcal{C}$ other than $\mathcal{C}$ itself exists. Therefore $\mathcal{C}$ satisfies Property 3.

We now show that all clusters satisfy Property 4 after the merge operation. By assumption, $\mathcal{C}'$ and clusters of Γ satisfy Property 4 before `merge`. From the algorithm, $\mathcal{C}'$ and all clusters in Γ are prefixed with $\mathcal{C}$. If the last bit of $\mathcal{C}'$ is 0, then by the distance definition $\mathcal{C}$ is closer to $\mathcal{C}'$ than to any cluster in Γ. In constrast, if the last bit of $\mathcal{C}'$ is 1 then the $(d'-1)^{th}$ neighbour of $\mathcal{C}'$ is the closest to $\mathcal{C}''$. Thus, after `merge`, $\mathcal{C}$ satisfies Property 4. Finally, since $\mathcal{C}'$ and all clusters in Γ are prefixed with $\mathcal{C}$, each cluster that referred to $\mathcal{C}'$ or to a cluster in Γ still refers to $\mathcal{C}$. This ends the proof. □

Lemma 12. *The number of messages incurred by a `merge` is* $\Theta(logN)$.

Proof. Given a core member p of a merging cluster $\mathcal{C}' = b_0 \dots b_{d-1} \dots b_{d'-1}$. A `merge` consists in contacting all the clusters prefixed with $b_0 \dots b_{d-1} \dots \overline{b}_{d'-1}$, with d the new dimension, and merging with them.

Thus by the same reasoning as previously, only a constant number of messages are exchanged by `commonprefix` to crawl the appropriate clusters. To merge the clusters, all the peers of each cluster are contacted. Thus the number of messages consumed is $\Theta(logN)$. Routing tables are updated by copying the routing tables of the appropriate cluster. Thus the routing tables updates is constant w.h.p.

As a consequence, the number of messages incurred by a `merge` is $\Theta(logN)$.

□

4.4.6 `create` Operation

The `create` operation enables to create a new cluster whenever sufficiently enough peers that share a common prefix do not find a cluster that match their prefixes. This operation consists in defining the label and the dimension of the new cluster $\mathcal{N}$, in finding outgoing and incoming links for $\mathcal{N}$, and in transferring to $\mathcal{N}$ the closest keys to it. Specifically, when core member p in $\mathcal{C}$ detects that temporary peers in its cluster satisfy predicate `tempIsSplit()`, then p computes $\mathcal{N}$'s label as the shortest common prefix shared by those T_{split} temporary peers. Then p randomly chooses S_{min} peers among those T_{split} peers and run consensus with all the other core members of $\mathcal{C}$ to agree on $\mathcal{N}$'s core set; the other $T_{split} - S_{min}$ peers being $\mathcal{N}$ spare members. Remaining temporary peers in $\mathcal{C}$ that are closer to $\mathcal{N}$ than to $\mathcal{C}$ are moved to $\mathcal{N}$ as temporary peers. Similarly for keys and associated data. Regarding outgoing links, each core member of $\mathcal{N}$ creates a routing table containing d entries such that entry i is filled with the closest cluster to $\mathcal{N}^{\overline{i}}$. This is achieved through `lookup` invocations. This construction is encapsulated in the `createOutgoingLinks()` procedure in Figure 6. Finally, core members build their predecessor table by contacting all clusters that should point to $\mathcal{N}$. These clusters are returned by procedure `pred`($\mathcal{N}$). This procedure is very similar to the `commonPrefix` one except that all paths not connected to $\mathcal{N}$ are pruned from the search.Note that both `commonPrefix` and `pred` procedure are detailed in [1]. Each returned cluster updates its routing table accordingly.

Lemma 13. *The `create` operation does not violate Properties 3 and Property 4.*

Proof. Let $\mathcal{C}' = b_0 \dots b_{d'-1}$. From Theorem 2, $\mathcal{C}'$ is the closest cluster to p. Let d be the position of the first different bit between p and $\mathcal{C}'$, that is p is prefixed with $b_0 \dots \overline{b}_{d-1}$. Since $\mathcal{C}'$ is the closest cluster to p, no cluster prefixed with $b_0 \dots \overline{b}_{d-1} = \mathcal{C}$ exists. Thus $\mathcal{C}$ satisfies Property 3.

From Theorem 2, the i^{th} neighbor of $\mathcal{C}$ is the closest cluster to $\mathcal{C}^{\overline{i}}$ for each $0 \leq i \leq d-1$. Thus $\mathcal{C}$ satisfies Property 4. Finally, since `pred` returns all the clusters that should have

`create`($\mathcal{C}$) /* run by core member p in the d-cluster $\mathcal{C}$*/

$\mathcal{T} \leftarrow$ set of the T_{split} *temporary* peers that share a prefix;
$\mathcal{N}$.`label` $\leftarrow$ shortest common prefix of $\mathcal{T}$'s members;
$\mathcal{N}$.`dim` $\leftarrow$ length of $\mathcal{N}.label$;
$V_c \leftarrow S_{min}$ randomly chosen peers within $\mathcal{T}$;
$V_c^{\mathcal{N}} \leftarrow$ run consensus on V_c among core members of $\mathcal{C}$;
$V_s^{\mathcal{N}} \leftarrow \{q \mid q \in \mathcal{T} \setminus V_c^{\mathcal{N}}\}$;
$\mathcal{N}$.`temp` $\leftarrow \{q \mid \mathcal{D}(q,\mathcal{N}) < \mathcal{D}(q,\mathcal{C})\}$;
p.`pred`($\mathcal{N}$);
p.`createOutgoingLinks()`;
enddo

Figure 6: `create` Operation at Peer p

outgoing links to $\mathcal{C}$, after `create`, all clusters other than $\mathcal{C}$ satisfy Property 4. This completes the proof. □

Lemma 14. *The number of messages incurred by the* `create` *operation is in* $\mathcal{O}(log^2 N)$.

Proof. Given a new cluster $\mathcal{C}$. A `create` consists in updating the routing table of $\mathcal{C}$ and of the concerned clusters. First, for each $i = 1 \ldots d$, the i^{th} entry of the routing table of $\mathcal{C}$ is updated to refer to the cluster found through `lookup`($\mathcal{C}^{\bar{i}}$). This operation costs in sum $O(log^2 N)$ messages. Second, clusters that must refer to $\mathcal{C}$ are contacted through `pred` which consumes $O(logN)$ messages. Then the number of messages incurred by a `create` is $O(log^2 N)$. □

Theorem 1. *Suppose that at some time t all the clusters of the system satisfy both Properties 3 and 4. Then, invocation of operations* `split`, `merge`, *or* `create` *does not jeopardize any of these properties.*

Proof. The theorem follows directly from Lemmata 6, 7, 12, and 13. □

4.4.7 Bootstrapping PeerCube

Initially the system contains S_{min} well known peers grouped together in a bootstrap cluster. That cluster has no label. We assume that no more than a fraction μ of these peers are Byzantine. When a new peer p wishes to join PeerCube it sends its joining request to one of those bootstrap peers[5]. Upon receipt of such a request, a bootstrap peer forwards it to all other bootstrap peers, acknowledges p, and inserts p in a waiting list. Peer p knows that its request has been accepted whenever it receives acks from $\lfloor \frac{S_{min}-1}{3} \rfloor + 1$ bootstrap peers. When the size of the bootstrap cluster (i.e., number of bootstrap peers plus number of peers in the waiting list) is such that there are at least *i*) S_{min} peers whose identifiers are prefixed by "0", and *ii*) S_{min} peers whose identifiers are prefixed by "1", then the bootstrap cluster splits into two new clusters $\mathcal{C}$ and $\mathcal{C}'$. $\mathcal{C}$ (resp. $\mathcal{C}'$) contains all the peers whose id is prefixed by "0" (resp. prefixed by "1"). Label of $\mathcal{C}$ (resp. $\mathcal{C}'$) is equal to the common prefix shared by its members, typically "0" (resp. "1"). Peers *state* is build: each $p \in \mathcal{C}$ creates its routing and predecessor tables, such that they both point to peers in $\mathcal{C}'$. The same applies for peers in $\mathcal{C}'$.

5 Handling Collusion

5.1 Thwarting Eclipse Attacks

An eclipse attack enables the adversary to control part of the overlay traffic by coordinating its attack to infiltrate routing tables of correct peers. As shown in the previous section,

[5]Note that we assume that p knows a correct peer. Otherwise there is no guarantee that p may ever join the system. This is a classic assumption in P2P overlays

PeerCube operations thwart those attacks essentially by preventing colluders from devising deterministic strategies to join core sets (i.e., newcomers are inserted as spare members) and by reaching agreement among core members on any event that affects PeerCube topology. Correctness of these operations relies on the hypothesis that no more than $\lfloor \frac{S_{min}-1}{3} \rfloor$ malicious peers populate core sets, that is the fraction of malicious peers in any core set is no more than 1/4. Probability that such an assumption does not hold is now discussed. Let us first compute the upper bound on the probability to corrupt a core set. This holds when the clusters number is minimal (i.e. equal to $\frac{N}{S_{max}}$). Denote by X_u the random variable describing the number of malicious peers in a cluster, and by Y_u the random variable describing the number of malicious peers in a core. Clearly, Y_u depends on X_u. Since identifiers are randomly chosen, inserting malicious peers into clusters can be interpreted as throwing $\mu.N$ balls one by one and randomly into $\frac{N}{S_{max}}$ bins. The probability that x balls (malicious peers) are inserted into a bin (cluster) is $P(X_u = x) = \binom{\mu.N}{x} \left(\frac{S_{max}}{N}\right)^x \left(1 - \frac{S_{max}}{N}\right)^{\mu.N-x}$. By the `leave` operation, each departure from a core set is followed by the rebuilding of this set with S_{min} randomly chosen peers among the S_{max} peers of the cluster. This can be interpreted as picking simultaneously S_{min} balls among S_{max} balls among which x are black (malicious peers) and $S_{max} - x$ are white (correct ones). Thus, the probability of having y malicious peers inserted in the core, knowing the number of malicious peers x in the cluster, is given by $P(Y_u = y | X_u = x) = \frac{\binom{x}{y}\binom{S_{max}-x}{S_{min}-y}}{\binom{S_{max}}{S_{min}}}$. Finally, the tight upper bound on the corruption probability is equal to

$$p_u = 1 - \sum_{y=0}^{\lfloor \frac{S_{min}-1}{3} \rfloor} \sum_{x=0}^{\mu.N} P(Y_u = y | X_u = x) P(X_u = x)$$

We now compute the lower bound on the corruption probability. This holds when the number of clusters in PeerCube is maximal (i.e. N/S_{min}), and thus clusters' population is minimal. Let X_l be the random variable representing the number of malicious peers in a cluster. Remark that X_l also represents the number of malicious peers in the core set. By proceeding as above, $P(X_l = x) = \binom{\mu.N}{x} \left(\frac{S_{min}}{N}\right)^x \left(1 - \frac{S_{min}}{N}\right)^{\mu.N-x}$. Thus, the tight lower bound on the corruption probability is

$$p_l = 1 - \sum_{x=0}^{\lfloor \frac{S_{min}-1}{3} \rfloor} P(X_l = x)$$

We can now derive upper and lower bounds on the probability that a request reaches its legitimate destination. The probability that the number of hops of a request be h is equal to $\binom{d_{max}}{h} \left(\frac{1}{2}\right)^{d_{max}}$, with $d_{max} = log_2(\frac{N}{S_{max}}) + 3$ the maximal dimension of a cluster. Such a request is successful if none of the h clusters crossed by this request are corrupted. Thus its probability of success is at least

$$\sum_{h=0}^{d_{max}} \binom{d_{max}}{h} \left(\frac{1}{2}\right)^{d_{max}} (1 - p_u)^h$$

and at most

$$\sum_{h=0}^{d_{min}} \binom{d_{min}}{h} \left(\frac{1}{2}\right)^{d_{min}} (1 - p_l)^h$$

with d_{min}, the minimum dimension of a cluster, is equal to $log_2(\frac{N}{S_{max}})$.

Recall that the policy we propose to replace a left core member is to refresh the whole composition of the core set by randomly choosing peers within the cluster. We opposed this policy to the one which consists in replacing the core member that left by a single one randomly chosen in the cluster (see Section 4.4.3). Figure 7 compares the lower bound on the probability of successful requests with these two policies according to S_{max}, for different

ratio of malicious peers in the system, and considering $N = 1,000$. The first observation is that probability of success for the policy we propose (labelled by `with randomisation` in the Figure) varies lightly with S_{max} value. This confirms that setting $S_{max} > \mathcal{O}(logN)$ does not bring any additional robustness to PeerCube. The second observation is that for the second policy (denoted by `w/o randomisation`), that probability drastically decreases with increasing values of S_{max}, even for small values of μ. This corroborates the weakness of such a policy in presence of a strong adversary.

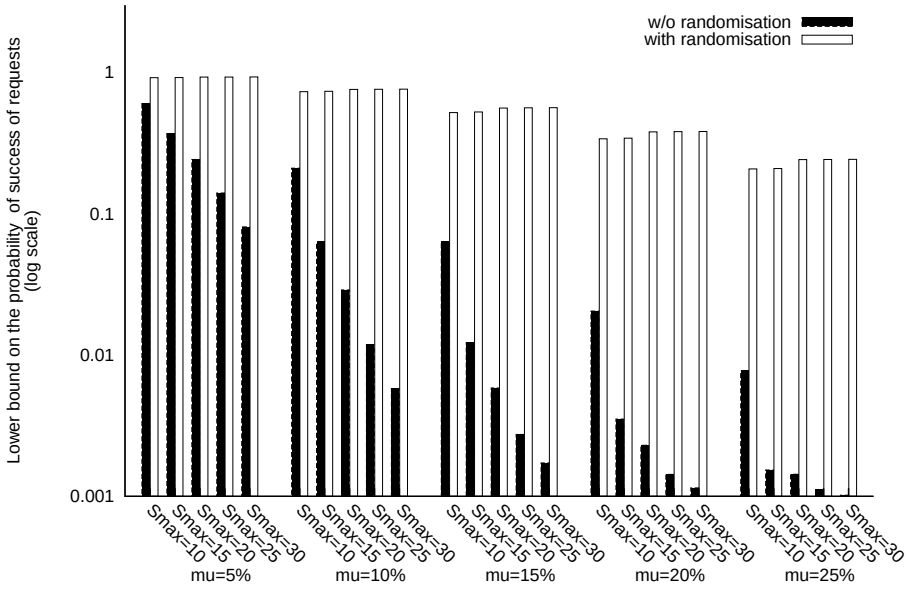


Figure 7: Probability of success of requests w.r.t. S_{max}

5.2 Robust Routing through Independent Routes

We have just seen that because identifiers are randomly assigned, the ratio of malicious peers in some clusters may exceed the assumed ratio μ of malicious peers in the system, (note that by our insertion algorithm, we guarantee that in expectation, the ratio of malicious peers in core sets is equal to the one in clusters), and thus may impact the resilience of PeerCube. Since pollution decreases with S_{min} (more precisely with $\lfloor (S_{min} - 1)/3 \rfloor$) a possible solution to increase the resilience is to augment S_{min} according to S_{max}, i.e. to have S_{min} in $\mathcal{O}(logN)$. However, because of the Byzantine resistant consensus this makes maintenance operations cost in $\mathcal{O}(logN^3)$ or in $\mathcal{O}(logN^2)$ because of the broadcast primitive. To circumvent this issue, we extend Castro et al. [3] approach by sending a request over independent routes. We adapt the independent routes construction algorithm presented in Section 4.1 to match PeerCube features. Essentially, the search is adapted to find the closest cluster to the theoretical one when this latter one does not exist. Denote by b the number of bit differences between p's identifier, the source of the request, and q's identifier, the destination peer. Recall that the i^{th} route is obtained by successively correcting bits $p_i, p_{i+1}, \ldots, p_{(i+b-1) \bmod b}$ for $0 \leq i \leq b-1$, with $p_i, p_{i+1}, \ldots, p_{(i+b-1) \bmod b}$ the position of the b bits that differ between p and q. We modify this procedure by invoking the `lookup` operation on keys obtained by successively correcting bits $p_i, p_{i+1}, \ldots, p_{(i+b-1) \bmod b}$ for $0 \leq i \leq b-1$. Other independent routes of non-optimal length are found by modifying first, one bit on which p and q both agree (say n_i), by looking for the closest cluster to that key, then by finding independent routes from that cluster by proceeding as above, and finally by re-modifying n_i.

Lemma 15. *The independent routes algorithm finds at least* $log_2 \frac{N}{S_{max}}$ *independent routes of length* $\mathcal{O}(logN)$ *w.h.p.*

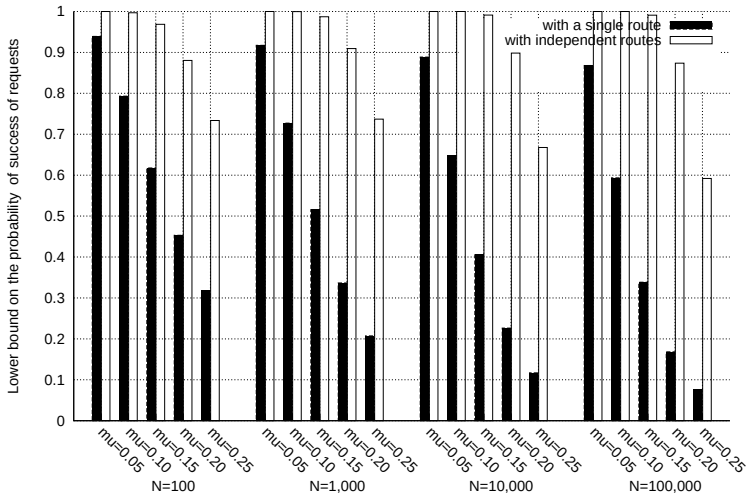


Figure 8: Probability of success of requests

Proof. Consider a d-cluster $\mathcal{C} = b_0 \dots b_{d-1}$. Consider the minimal dimension d_{min} of clusters. Then all the clusters are labelled with a prefix of length at least d_{min}; that is for all the labels of the clusters, at least d_{min} bits positions are always fully represented. Denote such bits positions by $\mathcal{P}$ ($card(\mathcal{P}) = d_{min}$). Then for some $k \in \mathcal{P}$, each $\texttt{lookup}(\mathcal{C}^{\overline{k}})$ invocation necessarily returns a cluster $\mathcal{C}_k$ such that its k^{th} bit is $\overline{b}_k$ and its i^{th} bit is b_i for all $i \in \mathcal{P}$. Thus the algorithm can be reduced to the standard independent routes construction in a perfect (d_{min})-hypercube. That is the number of independent routes is at least d_{min} which is greater than $log_2 N - log_2 S_{max}$.

Second, note that each $\texttt{lookup}(\mathcal{C}^{\overline{k}})$ procedure call in the independent routes algorithm incurs more than 1 hop only when the dimension of the closest cluster to $\mathcal{C}^{\overline{k}}$ is greater than the dimension of the current cluster (to correct the remaining bits). Indeed, if that is not the case, the closest cluster to $\mathcal{C}^{\overline{k}}$ is simply the k^{th} neighbour of the current cluster (this can be seen by recurrence). Consequently, since the maximum dimension difference is w.h.p. equal to 8, each `lookup` operation call in the algorithm consumes $O(1)$ number of messages. That is w.h.p. each independent route consumes $O(logN)$ messages. □

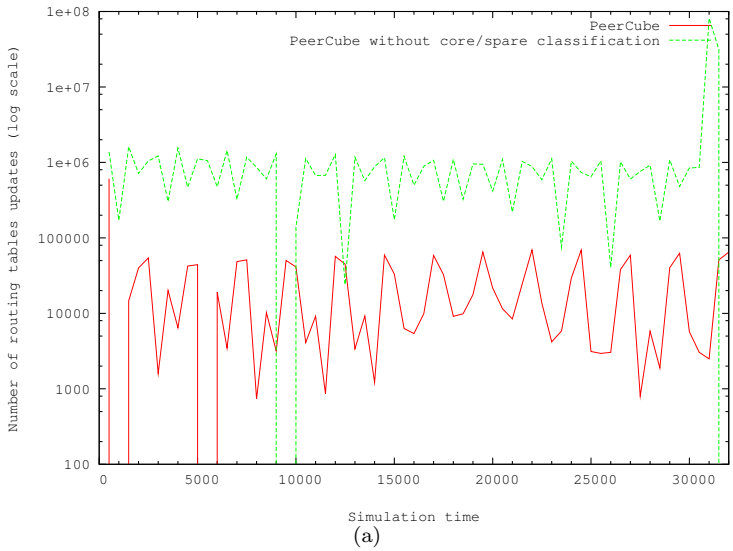
We now examine the probability p_{succ} for a request issued by a correct peer to reach its legitimate destination when that request is sent over r independent routes of length h, with $d_{min} \leq r \leq d_{max}$. The request is successful if at least one route does not contain any corrupted cluster. Let p denote the exact probability that a cluster is corrupted, i.e. $p_l \leq p \leq p_u$. The probability of success of a request using r independent routes of length h is $1 - \left(1 - (1-p)^h\right)^r$. Thus probability p_{succ} is lower bounded by

$$\sum_{h=0}^{d_{max}+2} \binom{d_{max}+2}{h} \left(\frac{1}{2}\right)^{d_{max}+2} \left(1 - \left(1 - (1-p_u)^h\right)^r\right)$$

and upper bounded by

$$\sum_{h=0}^{d_{min}} \binom{d_{min}}{h} \left(\frac{1}{2}\right)^{d_{min}} \left(1 - \left(1 - (1-p_l)^h\right)^r\right)$$

Term $d_{max}+2$ in the first equation comes from the non-optimal paths of the independent routing algorithm. Figure 8 shows the remarkable increase in PeerCube robustness when using independent routes w.r.t. to a single route. Note also that whatever the percentage of



(a)

Figure 9: Benefit of hot spares in PeerCube is displayed through the number of routing tables updates

malicious peers in the system the probability of success degrades gracefully (logarithmically) with respect to N.

6 Simulation

In this section, we present the results of an experimental evaluation of PeerCube performed on PeerSim a simulation platform for P2P protocols. The simulation is event based. The workload is characterised by the number of and arrival/departure pattern of peers and by the distribution of requests they issue. Each experiment uses a different workload.

Churn Impact In these experiments, we study the ability of PeerCube to greatly reduce the impact of high dynamics on peers load. In particular, we analyse the benefit drawn from appointing newcomers as spare members on the number of routing tables updates. In Figure 6, the number of routing tables updates in a network of up to 10,000 peers is depicted. Bursts of joins and leave are cyclically generated. $Smax = 13$, and $Smin = 4$. A failure-free environment is assumed. The dotted curve shows the number of triggered routing tables updates in a cluster-based hypercubic topology in which all clusters members actively participate in the overlay operations (denoted by `PeerCube without core/spare classification` in the figure), while the solid curve depicts the number of routing tables updates generated in PeerCube (denoted by `PeerCube`). As expected, using newcomers as hot spares drastically reduces the number of routing tables updates for both joins and departures events. For instance, the burst of joins generated during simulation time 27,000 and 27500 have triggered no routing tables updates for PeerCube while it has given rise to 50,400 updates for PeerCube without core/spare classification.

Robustness against Collusion In these experiments, we test the ability of PeerCube to achieve a robust lookup operation despite the presence of a strong adversary. As described in the previous section, robust lookup is realized by two techniques. First, by preventing malicious peers from strategizing to get inserted within core sets; through the randomization insertion algorithm, we minimize the ratio of malicious peers into routing tables. Second, by taking advantage of independent and optimal length paths offered by the hypercubic topology to guarantee that a request sent by a correct peer reaches its legitimate destination with probability close to 1. Figure 10 shows for $N = 1,000$ peers, the probability of successful

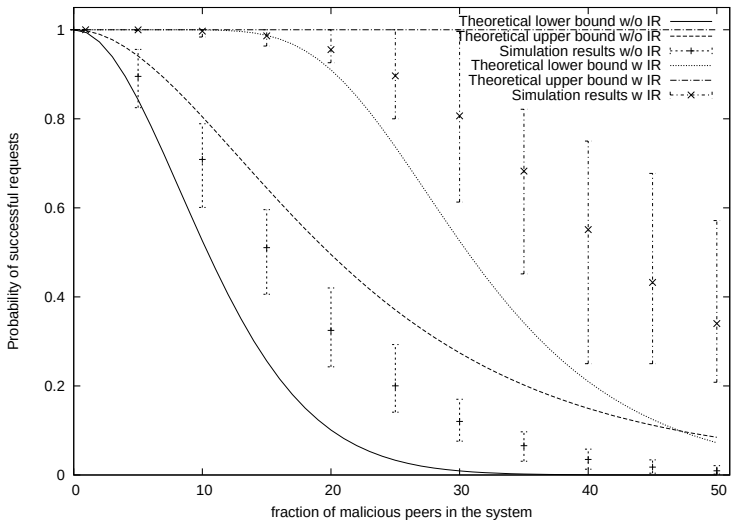


Figure 10: Probability of success wrt malicious peers

requests sent by correct peers w.r.t. to the ratio of malicious peers in the system. The main observation is that experiments fully validate theoretical results. Namely, for up to 15% of malicious peers, 98% of the requests issued from correct peers are successful, and for 25% of malicious peers, in average, 90% of the requests are successful, which clearly emphasises PeerCube robustness to co-ordinated malicious behaviour.

7 Conclusion

In this paper we have presented PeerCube, a DHT-based system that is able to handle high churn and collusive behavior. Many existing P2P systems exhibit some fault tolerance or churn resiliency. The main contribution of PeerCube is to combine existing techniques from classical distributed computing and open large distributed systems in a new way to efficiently decrease churn impact and to tolerate collusion of malicious peers as shown analytically and validated through experimental simulation. For future work, we are planning to study strategies against a computationally unbounded adversary, that is an adversary, beyond being able to inspect the whole system and issue join and leave requests as often it wishes (as studied in this paper), can carefully choose the IDs of the Byzantine peers, so that it can place them at critical locations in the network [7, 20].

References

[1] E. Anceaume, F. Brasiliero, R. Ludinard, and A. Ravoaja. Peercube: an hypercube-based p2p overlay robust against collusion and churn. Technical Report 1888, IRISA, 2008.

[2] G. Bracha. An asynchronous [(n-1)/3]-resilient consensus protocol. In *Proc. of the ACM Symposium on Principles of Distributed Computing (PODC)*, 1984.

[3] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *Proc. of the Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[4] M. Correia, N. Ferreira Neves, and P. Verissimo. From consensus to atomic broadcast: Time-free byzantine-resistant protocols without signatures. *Computer Journal*, 49(1), 2006.

[5] D. Dolev, E. Hoch, and R. van Renesse. Self-stabilizing and byzantine-tolerant overlay network. In *Proc. of the Int'l Conference On Principles Of Distributed Systems (OPODIS)*. LNCS 4878, 2007.

[6] J. Douceur. The sybil attack. In *Proc. of the Int'l Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[7] A. Fiat, J. Saia, and M. Young. Making chord robust to byzantine attacks. In *Proc. of the Annual European Symposium on Algorithms (ESA)*, 2005.

[8] J.A. Garay and Y. Moses. Fully polynomial byzantine agreement for $n > 3t$ processes in $t + 1$ rounds. *SIAM Journal on Computing*, 27(1), 1998.

[9] K. Hildrum, J.Kubiatowicz, S.Rao, and B.Zhao. Distributed data location in a dynamic network. In *Proc. for the ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, 2002.

[10] K. Hildrum and J. Kubiatowicz. Asymptotically efficient approaches to fault-tolerance in peer-to-peer networks. In *Proc. of the Int'l Symposium on Distributed Computing (DISC)*, 2003.

[11] J. Li, J. Stribling, T. Gil, R. Morris, and F. Kaashoek. Comparing the performance of distributed hash tables under churn. In *Proc. for the Int'l Workshop on Peer-to-Peer Systems (IPTPS)*, 2004.

[12] T. Locher, S. Schmid, and R. Wattenhofer. eQuus: A provably robust and locality-aware peer-to-peer system. In *Proc. of the Int'l Conference on Peer-to-Peer Computing (P2P)*, 2006.

[13] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Proc. for the Int'l Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[14] H. Ramasamy and C. Cachin. Parsimonious asynchronous byzantine-fault-tolerant atomic broadcast. In *Proc. of the Int'l Conference On Principles Of Distributed Systems (OPODIS)*. LNCS 3974, 2005.

[15] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. of the ACM SIGCOMM*, 2001.

[16] A. Ravoaja and E. Anceaume. Storm: A secure overlay for p2p reputation management. In *Proc. of the Int'l IEEE conference on Self-Autonomous and Self-Organizing Systems (SASO)*, 2007.

[17] R. Rivest. The md5 message digest algorithm, 1992.

[18] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of the Int'l Conference on Distributed Systems Platforms (Middleware)*, 2001.

[19] Y. Saad and M. Schultz. Topological properties of hypercubes. *IEEE Transactions on Computers*, 37(7), 1988.

[20] C. Scheideler. Robust random number generation for peer-to-peer systems. In *Proc. of the Int'l Conference On Principles Of Distributed Systems (OPODIS)*. LNCS 4305, 2006.

[21] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables. In *Proc. of the Int'l Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[22] M. Srivatsa and L. Liu. Vulnerabilities and security threats in structured peer-to-peer systems: A quantitiative analysis. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*, 2004.

[23] I. Stoica, D. Liben-Nowell, R. Morris, D. Karger, F. Dabek, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. of the ACM SIGCOMM*, 2001.

[24] A.C. Yao. Probabilistic computations: toward a unified measure of complexity. In *Proc. of the IEEE Symp. on Foundations of Computer Science (FOCS)*, 1977.

Appendix

8 Dimensions Disparity in PeerCube

The next lemmata give (probabilistic) bounds on the dimensions of clusters according to the maximal size S_{max} of the clusters.

Lemma 16. *The dimension of a cluster is greater than $log_2 N - log_2 S_{max}$.*

Proof. The minimum dimension is given by the minimum number of bits needed to code the labels of the clusters. Note that the lower the number of clusters, the lower the dimensions of these clusters (the lower the minimum number of bits needed to code them). Since the maximum number of peers per cluster is S_{max}, the minimum number of clusters is N/S_{max}. Thus, the minimum number of bits needed to code the label of a cluster is $log_2 N - log_2 S_{max}$. □

Lemma 17. *If $S_{max} \geq log_2 N$, w.h.p. the dimension of a cluster is lower than $log_2 N - log_2 S_{max} + 3$.*

Proof. Let D be the random variable representing the dimension of a cluster, and X_d be the random variable that represents the number of peers suffixed with a bit string of length d. Then if the dimension of the cluster is greater than d, the number of peers suffixed with the string of length d is greater than S_{max}. Thus $Prob(D > d) \leq Prob(X_d > S_{max})$.

By assumption, peers identifiers are assigned uniformly randomly. Thus X_d follows a binomial law of parameters N and $\frac{1}{2^d}$ where $\frac{1}{2^d}$ is the probability that a peer is suffixed with a string of length d. Thus by the Chernoff's upper bound, for $S_{max} > \frac{N}{2^d}$:

$$Prob(D > d) \leq \left(\frac{e^{S_{max}\frac{2^d}{N}-1}}{\left(\frac{2^d}{N}S_{max}\right)^{\frac{2^d}{N}S_{max}}} \right)^{\frac{N}{2^d}}$$

$$\leq \frac{e^{S_{max}-\frac{N}{2^d}}}{\left(\frac{2^d}{N}S_{max}\right)^{S_{max}}} \leq \frac{e^{S_{max}}}{\left(\frac{2^d}{N}S_{max}\right)^{S_{max}}}.$$

If we suppose that $\frac{2^d}{N}S_{max} \geq 2e$, then $Prob(D > d) \leq \frac{1}{2^{S_{max}}}$. It follows that, if $d > log_2 N - log_2 S_{max} + log_2(2e)$ and if $S_{max} \geq log_2 N$, then we have $Prob(D > d) \leq \frac{1}{N}$. □

This leads to the following proposition:

Proposition 1. *Let δ be the difference between the dimensions of any two clusters. If $S_{max} \geq log_2 N$, then $2^\delta \leq 8$ w.h.p.*

Proof. This follows from lemmata 17 and 16. □

Proposition 2. *If $S_{max} \geq log_2 N$, the number of non-represented prefixes is w.h.p. at most 8.*

Proof. Since w.h.p the dimension of a cluster is lower than $log_2 N - log_2 S_{max} + 3$, the number of non-represented prefixes is maximal when the dimensions of the clusters are all $log_2 N - log_2 S_{max} + 3$ and the number of clusters is minimal (N/S_{max}). Since the minimal number of bits needed to code N/S_{max} clusters is $log_2 N - log_2 S_{max}$, the number of non-represented prefixes is at most $2^3 = 8$. □

9 `commonprefix` and `pred` Procedures

9.1 `commonprefix` Procedure

The `commonprefix` procedure is used for contacting all the clusters prefixed with a given bit string $b_0 \dots b_i$. It guarantees that each cluster prefixed with $b_0 \dots b_i$ is crossed once and only once. This makes `commonprefix` optimal in messages number, i.e., the number of sent messages is equal to the number of clusters prefixed by $b_0 \dots b_i$. This procedure is based on a constrained flooding approach. The constraint avoids messages redundancy, and is derived from the property of routing tables (Lemma 1).

Suppose that some peer $p \in \mathcal{C}$ wishes to find all clusters that share a common prefix with $\mathcal{C}$. Let $b_0 \dots b_i$ be that common prefix. Let $d' \geq i+1$ be $\mathcal{C}$'s dimension. Then p proceeds as follows. For $k = i+1, \dots, d'-1$, p sends a `commonprefix`$(b_0 \dots b_i, k)$ message to an arbitrary core member of the k^{th} entry of its routing table. When a core member q receives such a request, it sends back to p his identifier, and proceeds as above by forwarding this request to a random core member of the k'^{th} entry of its routing table with $k' = k+1, \dots, d''-1$, with d'' the dimension of q's cluster. If $d'' = k+1$, q does not forward the request to its neighbours.

Lemma 18. *An invocation of `commonprefix`$(b_0 \dots b_i)$ contacts once and only once all clusters prefixed by $b_0 \dots b_i$.*

Proof. Let $\mathcal{C}_0$ be a d-cluster $b_0 \dots b_i b_{i+1} \dots b_{d-1}$ initiating a `commonprefix`$(b_0 \dots b_i)$ procedure call. We first show that all clusters prefixed by $b_0 \dots b_i$ are contacted. Consider a d'-cluster $\mathcal{C}' = b_0 \dots b_i a_{i+1} \dots a_{d-1}$. By the algorithm and by Property 4, each bits b_j, for $j > i$ of $\mathcal{C}_0$ is eventually flipped to a_j. Since a corrected bit is never flipped again $\mathcal{C}'$ is eventually contacted.

We now prove that clusters are contacted only once. Briefly, the proof consists in showing that if a cluster is contacted by two different clusters then this cluster has two incompatible prefixes (i.e., none of them is a prefix of the other one).

We proceed by contradiction. Suppose that some cluster $\mathcal{C}'$ prefixed by $b_0 \dots b_i$ is contacted by two different clusters $\mathcal{A}_j$ and $\mathcal{B}_k$. Then it must be the case that two routes P_1 and P_2 exist between $\mathcal{C}$ and $\mathcal{C}'$ with $P_1 = \mathcal{C}_0 \dots \mathcal{C}_l \mathcal{A}_0 \dots \mathcal{A}_j \mathcal{C}'$ and $P_2 = \mathcal{C}_0 \dots \mathcal{C}_l \mathcal{B}_0 \dots \mathcal{B}_k \mathcal{C}'$.

Note that $\mathcal{C}_0$ is not necessarily different from $\mathcal{C}_l$. Suppose that $\mathcal{A}_0$ (resp. $\mathcal{B}_0$) is the i'^{th} (resp. i''^{th}) neighbour of $\mathcal{C}_l$ (i.e., bit i' is the first bit that differs from $\mathcal{A}_0$ and $\mathcal{C}_l$.). Thus, from Lemma 1, the i'^{th} bit of $\mathcal{A}_0$ is different from the i'^{th} bit of $\mathcal{B}_0$. From the algorithm, all the clusters traversed from $\mathcal{A}_0$ to $\mathcal{C}'$ must share the same prefix of length $i'+1$, i.e., the prefix $b_0 \dots b_i \dots \overline{a}_{i'}$. Similarly for all the clusters traversed from $\mathcal{B}_0$ to $\mathcal{C}'$ that must share the same prefix of length $i''+1$, i.e., prefix $b_0 \dots b_i \dots a_{i'} \dots \overline{a}_{i''}$. Which is impossible as $\mathcal{C}'$ cannot share both prefix $b_0 \dots b_i \dots \overline{a}_{i'}$ and prefix $b_0 \dots b_i \dots a_{i'} \dots \overline{a}_{i''}$. This concludes the proof. □

Lemma 19. *Let $b_0 \dots b_{d-1}$ some cluster label. The number of messages incurred by `commonprefix`$(b_0 \dots b_{d-1})$ is constant w.h.p.*

Proof. By Lemma 18, `commonprefix` crosses each cluster prefixed with $b_0 \dots b_{d-1}$ exactly once. Let $\mathcal{C}$ be the cluster prefixed with $b_0 \dots b_{d-1}$ of which the dimension is the greatest one among all clusters prefixed with $b_0 \dots b_{d-1}$. Let δ be the difference between $\mathcal{C}$'s dimension and d. Then the number of clusters crawled by `commonprefix`$(b_0 \dots b_{d-1}$,-) is at most 2^δ. Thus, if d represents the dimension of a cluster labelled $b_0 \dots b_{d-1}$, then by Proposition 1 the number of clusters crawled by `commonprefix` is $O(1)$ w.h.p. Finally, since only core members are involved in `commonprefix` and since the number of core peers per cluster is constant, the number of messages incurred by `commonprefix` is w.h.p. constant. □

9.2 `pred` Procedure

We now describe how peers of a given d-cluster $\mathcal{C}$ locate all clusters that have outgoing links to $\mathcal{C}$ through the `pred` procedure. The procedure is similar to `commonprefix` except that all

routes not connected to $\mathcal{C}$ are pruned. Suppose that some peer $p \in \mathcal{C}$ wishes to locate all the clusters that have an outgoing link to $\mathcal{C}$. Then p proceeds as follows:

1. For each $0 \leq i \leq d-1$, p sends a `pred`($\mathcal{C}$, $\mathcal{E}$,i) message to its i^{th} neighbour $\mathcal{C}_i$. The set $\mathcal{E}$ indicates the entries of p's routing table that refer to $\mathcal{C}$ itself.
2. If $\mathcal{C}_i$ has an outgoing link to $\mathcal{C}$, then $q \in \mathcal{C}_i$ sends a feedback message to $\mathcal{C}$. In all cases q forwards `pred`($\mathcal{C}$, $\mathcal{E}$, i') to its neighbor $r \in \mathcal{C}_{ii'}$ for each $i' > i$ such that either $i' \in \mathcal{E}$ or $i' > d$.
3. $r \in \mathcal{C}_{ii'}$ in turn executes Step 2 unless i' is equal to the dimension of $\mathcal{C}_{ii'}$.

We can state the following lemma.

Lemma 20. *Let $p \in \mathcal{C}$ such that p invokes `pred`(). Then all the clusters that have outgoing links to $\mathcal{C}$ are contacted once and only once.*

Proof. Note first that, in the algorithm, if the condition "$i' \in \mathcal{E}$ or $i' > d$" was removed, the algorithm would be equivalent to a `commonprefix` procedure on a prefix of length 0. By Lemma 18, all the clusters of the system would be contacted once and only once. To prove that all clusters that have outgoing links to $\mathcal{C}$ are contacted, it suffices to prove that all the clusters that do not satisfy this condition cannot have any outgoing link to $\mathcal{C}$.

First consider any two clusters $\mathcal{A}$ and $\mathcal{B}$, and denote by i the position of the first different bit between $\mathcal{A}$ and $\mathcal{B}$. Then $\mathcal{A}$ has an outgoing link to $\mathcal{B}$ if and only if $\mathcal{B}$ is the i^{th} neighbor of $\mathcal{A}$. Indeed by Lemma 4 for any $j \neq i$ the j^{th} neighbor of $\mathcal{A}$ (if it is not $\mathcal{A}$) differ with $\mathcal{B}$ by at least its $\min(i,j)^{th}$ bit and thus cannot be equal to $\mathcal{B}$. Thus, to prove that some cluster $\mathcal{A}$ cannot have any outgoing link to some cluster $\mathcal{B}$, it suffices to prove that $\mathcal{B}$ cannot be the i^{th} neighbor of $\mathcal{A}$ with i the position of the first different bit between $\mathcal{A}$ and $\mathcal{B}$.

Now given a cluster $\mathcal{C}'$ having received a `pred`($\mathcal{C}$,$\mathcal{E}$,i) message. Let us show that for each $i' > i$ s.t. $i' \notin \mathcal{E}$ the i'^{th} neighbor $\mathcal{C}'_{i'}$ of $\mathcal{C}'$ cannot have any outgoing link to $\mathcal{C}$. Note first that from the algorithm, the first different bit between $\mathcal{C}'_{i'}$ and $\mathcal{C}$ is some j, such that $j \leq i < i'$. For simplicity, we assume that $j = i$

- $\mathcal{C} = a_0 \dots a_i \dots a_{i'} \dots$,
- $\mathcal{C}' = a_0 \dots \overline{a}_i \dots a_{i'} \dots$,
- $\mathcal{C}'_{i'} = a_0 \dots \overline{a}_i \dots \overline{a}_{i'} \dots$.

But since $i' \notin \mathcal{E}$, the i'^{th} neighbor $\mathcal{C}_{i'}$ of $\mathcal{C}$ is prefixed by $a_0 \dots a_i \dots \overline{a}_{i'}$. Then we necessarily have,

$$\mathcal{D}(a_0 \dots \overline{a}_i \dots \overline{a}_{i'}, a_0 \dots a_i \dots \overline{a}_{i'}) < \mathcal{D}(a_0 \dots \overline{a}_i \dots \overline{a}_{i'}, a_0 \dots a_i \dots a_{i'})$$

That is,

$$\mathcal{D}(\mathcal{C}'^{\overline{i}}_{i'}, \mathcal{C}_{i'}) < \mathcal{D}(\mathcal{C}'^{\overline{i}}_{i'}, \mathcal{C})$$

Thus $\mathcal{C}$ is not the i^{th} neighbor of $\mathcal{C}'_{i'}$, with i the position of the first different bit between $\mathcal{C}'_{i'}$ and $\mathcal{C}$, and $\mathcal{C}'_{i'}$ cannot have any outgoing link to $\mathcal{C}$. □

Lemma 21. *The number of messages incurred by `pred` is w.h.p. in $\mathcal{O}(logN)$*

Proof. Consider a d-cluster $\mathcal{C}$ invoking `pred`. To prove the lemma, it suffices to observe first that the number of non-existing entries of $\mathcal{C}$'s routing table is w.h.p. at most 8. Second, the maximum difference between the dimensions of two clusters is w.h.p 3. Thus, each `pred` message sent to a neighor of $\mathcal{C}$, is forwarded at most O(1) times. Finally, since the number of neighbor of $\mathcal{C}$ is $O(logN)$, the number of messages consumed by a `pred` is $O(logN)$. □